Chapter 12 Safety engineering:

12.1 Safety-critical systems

12.2 Safety requirements

12.3 Safety engineering processes

12.4 Safety cases

Note: Images excluded due to time constraints

Tree view:

- 12.1 Safety-critical systems
- 12.2 Safety requirements
  - 12.2.1 Hazard Identification
  - 12.2.2 Hazard Assessment
  - 12.2.3 Hazard Analysis
  - 12.2.4 Risk Reduction
- 12.3 Safety engineering processes
  - 12.3.1 Safety Assurance Processes
  - 12.3.2 Formal Verification
  - 12.3.3 Model Checking
  - 12.3.4 Static Program Analysis
- 12.4 Safety cases
  - 12.4.1 Structured Arguments
  - 12.4.2 Software Safety Arguments

12.1 Safety-Critical Systems

Overview

- **Safety-critical systems** must operate without causing harm to people or the environment, irrespective of whether they conform to their specification.
- Examples include control systems in aircraft, chemical plants, and automobile control systems.

Classes of Safety-Critical Software

1. **Primary Safety-Critical Software**: Embedded as a controller in systems. Malfunction can cause immediate harm. E.g., insulin pumps.
2. **Secondary Safety-Critical Software**: Indirectly causes harm. E.g., engineering design systems, patient management systems.

Techniques for Safety

1. **Hazard Avoidance**: Design systems to avoid hazards, like requiring two-hand operation in a paper-cutting machine.
2. **Hazard Detection and Removal**: Systems detect and mitigate hazards before they result in accidents.
3. **Damage Limitation**: Systems include features that minimize damage from accidents, like fire extinguishers in aircraft engines.

Concept of Hazard

- A **hazard** is a system state that could lead to an accident but isn't an accident itself. Reducing hazards effectively reduces accidents.

Complexity and Accidents

- Systems can cope with single failures but struggle when multiple things go wrong simultaneously.
- Due to increasing complexity, it’s difficult to predict the consequences of a combination of unexpected system events.

Safety Terminology

- **Accident**: Unplanned events causing harm.
- **Damage**: Measure of loss from an accident.
- **Hazard Probability**: The likelihood of a hazard occurring.
- **Hazard Severity**: Assessment of worst possible damage from a hazard.
- **Risk**: Measure of the likelihood that a system will cause an accident.

Risk-Based Requirements Specification

- Focuses on events causing most damage or occurring frequently.
- Involves understanding risks, discovering root causes, and generating requirements to manage these risks.

Realistic Expectations

- 100% safety is impossible to achieve.
- Society has to weigh the benefits of advanced technologies against the risk of occasional accidents.

Understanding the complexities and challenges in safety-critical systems is crucial for developing systems that minimize harm while maximizing utility.

12.2 Safety Requirements

Overview

- **System safety** is not only about good engineering but also necessitates detailed attention in deriving system requirements.
- Safety requirements are primarily functional requirements aimed at ensuring system safety against failures and external attacks.

"Shall Not" Requirements

- Unlike usual functional requirements that specify what a system should do, **"shall not" requirements** outline what the system should avoid.
- Examples include not allowing reverse thrust in flight or limiting the activation of alarm signals.
- These "shall not" requirements are decomposed into more specific functional software requirements or addressed through system design decisions.

Sources for Generating Requirements

- The basis for generating functional safety requirements usually includes **domain knowledge, safety standards, and regulations**.

Balance between Safety and Functionality

- Safety requirements may call for system shutdown to maintain safety.
- There is a need to balance **safety and functionality** to avoid overprotection and ensure cost-effective operation.

Risk-Based Requirements Specification

- Focuses on identifying and mitigating risks. Particularly useful for safety-critical systems.

Hazard-Driven Safety Specification Process

1. **Hazard Identification**: Identifies potential hazards threatening the system, often recorded in a hazard register.
2. **Hazard Assessment**: Decides which hazards are most dangerous or likely to occur and prioritizes them for safety requirements.
3. **Hazard Analysis**: Conducts root-cause analysis to determine the events leading to each hazard.
4. **Risk Reduction**: Based on hazard analysis, identifies safety requirements either to prevent the hazard or to minimize damage if an accident does occur.

By understanding and incorporating these elements, it's possible to derive comprehensive and effective safety requirements for critical systems.

12.2.1 Hazard Identification

Overview

- **Hazard identification** is the starting point in safety-critical systems for establishing safety measures.
- This process involves multiple experts such as experienced engineers, domain experts, and professional safety advisers.

Types of Hazards

- Hazards are categorized into different classes like **physical, electrical, biological, radiation**, and **service failure hazards**.
- Each class is analyzed to identify specific hazards, including dangerous combinations of hazards.

Methods for Identifying Hazards

- **Group working techniques**, such as brainstorming, often used.
- In the case of an insulin pump system, involved experts could include doctors, medical physicists and engineers, and software designers.

Examples of Hazards in Insulin Pump System

- Insulin overdose computation (service failure)
- Insulin underdose computation (service failure)
- Failure of hardware monitoring system (service failure)
- Power failure due to exhausted battery (electrical)
- Electrical interference with other medical equipment (electrical)
- Poor sensor and actuator contact (physical)
- Parts of the machine breaking off in the patient's body (physical)
- Infection caused by the introduction of the machine (biological)
- Allergic reaction to materials or insulin (biological)

Software-Related Hazards

- Concerned mainly with **failure to deliver a system service** or **failure of monitoring and protection systems**.

Hazard Register

- A **hazard register** may be used to record identified hazards and reasons for their inclusion.
- Serves as an important legal document, useful in subsequent inquiries or legal proceedings to demonstrate due diligence in hazard analysis.

12.2.2 Hazard Assessment

Overview

- **Hazard assessment** aims to understand the factors leading to hazards and the consequences if an accident occurs.
- The process helps in managing risks and determining the seriousness of each hazard.

Risk Categories in Hazard Assessment

1. **Intolerable Risks**: Threaten human life and must be designed out of the system or detected before causing an accident.
    - Example: Overdose of insulin in an insulin pump system.
2. **As Low As Reasonably Practical (ALARP) Risks**: Serious but with a very low likelihood, or less severe but more likely. The system should minimize these risks.
    - Example: Failure of the hardware monitoring system in an insulin pump.
3. **Acceptable Risks**: Normally result in minor damage and should be reduced if it doesn't significantly impact cost or other factors.
    - Example: Risk of an allergic reaction in the insulin pump user.

Risk Triangle

- Illustrates the costs of ensuring that risks do not result in incidents or accidents.
- Boundaries between risk regions are not fixed and can vary based on societal attitudes.

Societal Influence

- Public opinion may demand money be spent on reducing the likelihood of a system accident, irrespective of cost.
- Risk classifications may change due to external events like terrorist attacks or natural phenomena.

Risk Classification for Insulin Pump

- Hazards are classified based on estimated probability and severity (e.g., Insulin overdose is high risk, power failure is acceptable risk).

Role of Software in Hazard Detection

- Monitoring software can warn of potential problems, allowing hazards to be detected before causing an accident.

Challenges

- Estimating hazard probability and risk severity is difficult due to the rarity of hazards and accidents.
- Engineers often use relative terms like "probable," "unlikely," "rare," "high," "medium," and "low" due to a lack of statistical data.

12.2.3 Hazard Analysis

Overview

- **Hazard analysis** is the process of identifying the root causes that could lead to hazards in a safety-critical system.
- Two main approaches exist: **top-down (deductive)** and **bottom-up (inductive)**.

Techniques for Hazard Analysis

- **Fault Tree Analysis**: A commonly used top-down technique for both hardware and software hazards.
  - Easy to understand without specialist domain knowledge.

Steps in Fault Tree Analysis

1. Start with identified hazards.
2. Work backward to find the possible causes of each hazard.
3. Decompose until root causes are found.

Example: Insulin Delivery System

- **Incorrect Insulin Dose Administered** is considered as a single hazard for simplification.
  - Three main conditions leading to incorrect dosage:
    a. Incorrect blood sugar level measurement.
    b. Delivery system failure.
    c. Timing issues in insulin delivery.

Fault Tree Branches for Insulin System

1. **Left Branch**: Concerns incorrect measurement of blood sugar level.
   - Causes: Sensor failure or incorrect computation.
   - Incorrect computation may result from an incorrect algorithm or arithmetic errors.
2. **Central Branch**: Concerns timing issues.
   - Causes: System timer failure.
3. **Right Branch**: Concerns delivery system failure.
   - Causes: Incorrect computation of insulin requirement or failure in signaling to the insulin pump.
   - Incorrect computation can result from algorithm failure or arithmetic errors.

Role of Hardware in Hazard Analysis

- Fault trees can also help in identifying hardware problems.
- Provides insights into requirements for software to detect and correct hardware problems.

Additional Insights

- Processor capacity in the insulin system can be used for diagnostic and self-checking programs.
- Hardware errors like sensor, pump, or timer failures can be discovered and warnings issued to prevent serious effects.

12.2.4 Risk Reduction

Overview

- **Risk Reduction** involves deriving safety requirements that manage identified risks and hazards, aiming to prevent incidents or accidents.

Strategies for Risk Reduction

1. **Hazard Avoidance**: Design the system so that the hazard cannot occur.
2. **Hazard Detection and Removal**: Design the system to detect and neutralize hazards before they lead to an accident.
3. **Damage Limitation**: Minimize the consequences of an accident if it does occur.

Mixed Approaches

- Typically, designers use a **combination of these strategies**.
- In safety-critical systems, intolerable hazards are often mitigated by both minimizing their probability and adding a backup protection system.

Example: Insulin Delivery System

- A safe state is a shutdown state where no insulin is injected.
- Solutions for software failures leading to incorrect dosage:
  i. **Arithmetic Error**: Identify all potential arithmetic errors and include exception handlers for each.
     - Default action: Shut down the delivery system and activate a warning alarm.
  ii. **Algorithmic Error**: No clear program exception; can be detected by comparing doses.
     - Actions: Issue warnings and limit further dosage if a sequence of above-average doses has been delivered.

Sample Safety Requirements (Figure 12.6)

- **SR1**: Maximum limit for a single dose of insulin.
- **SR2**: Maximum limit for daily cumulative dose.
- **SR3**: Hardware diagnostic to be executed at least four times per hour.
- **SR4**: Exception handlers for all identified exceptions.
- **SR5**: Audible alarm and diagnostic message for any hardware or software anomaly.
- **SR6**: Suspend insulin delivery in the event of an alarm until user reset.

Note: These are user requirements and would be detailed further in the system requirements specification.

12.3 Safety Engineering Processes

Overview

- **Safety Engineering Processes** in safety-critical software development are an extension of software reliability engineering.
- Typically follow a **plan-based, waterfall model** with rigorous reviews and checks at each stage.
- Focus on **fault avoidance and fault detection**.

Importance of Reliability and Additional Verification

- **Reliability** is a prerequisite but not sufficient; additional **verification activities** may include formal models, static analysis tools, and other methods to discover potential faults.

Safety Reviews

- **Safety reviews** involve engineers and stakeholders to explicitly look for potential safety issues.

Regulatory Requirements

- Some systems are regulated by **national and international bodies**.
- Required evidence includes:
  i. **System Specification and Checks**: Detailed system specs and records of checks made on that specification.
  ii. **Verification and Validation**: Documentation on V&V processes and results.
  iii. **Organizational Processes**: Evidence of defined, dependable software processes that include safety assurance reviews.

Unregulated Systems

- For **unregulated systems**, like car-based systems, the responsibility lies with the manufacturer.
- Detailed safety information must be maintained to defend against **legal action**.

Documentation Necessity

- Extensive **process and product documentation** is mandatory for legal and regulatory reasons.

Limitations of Agile Methods

- **Agile processes** are not suitable without modification because they lack the required focus on extensive documentation.

Dependable Processes

- Safety-critical systems should be based on **dependable processes**, which include:
  - Requirements management
  - Change management and configuration control
  - System modeling
  - Reviews and inspections
  - Test planning and test coverage analysis
  - Additional safety assurance and verification analyses.

12.3.1 Safety Assurance Processes

Overview

- **Safety Assurance** is a comprehensive set of activities that ensures a system operates safely.

- Activities are thoroughly **documented** and may serve as evidence for regulatory compliance or system safety.

Examples of Safety Assurance Activities

1. **Hazard Analysis and Monitoring**: Traces hazards from preliminary analysis to testing and system validation.
2. **Safety Reviews**: Conducted throughout the development process to identify potential safety issues.
3. **Safety Certification**: Formal certification of the safety of critical components by an external group.

Role of Safety Engineers

- **Project Safety Engineers** are appointed with explicit responsibility for safety aspects and are accountable for any safety-related system failure.
- Work with **Quality Managers** to maintain detailed **Configuration Management** for safety-related documentation.

Hazard Register

- The **Hazard Register** is a central document that records how identified hazards are addressed during software development.
- Individuals responsible for safety are explicitly identified for accountability and legal reasons.

Safety Reviews

- Driven by the hazard register, these reviews focus on the system's ability to cope with identified hazards.
- Any issues identified must be addressed by the development team.

Software Safety Certification

- Required when integrating **external components** into safety-critical systems.
- Certification team performs extensive **verification and validation** and may require the external component developers to demonstrate use of dependable processes.

Licensing of Safety Engineers

- In some jurisdictions, safety engineers must be **licensed**, with a defined minimum level of qualifications and experience.

Importance of Documentation

- Documentation is essential for traceability, especially where **external certification** is required, such as in aircraft systems.

12.3.2 Formal Verification

Overview

- **Formal Methods** are used in safety-critical systems to mathematically analyze and verify the system's specification.
- They bridge the gap between comprehensive system testing and assurance of fault-free operation.

Drivers and Applications

- Need for assurance in safety-critical systems led to the development of formal methods.
- Used in various sectors like **railway systems** and by companies like **Airbus** for critical systems.

Stages in Formal Verification

1. **Formal Specification**: The system specification is mathematically analyzed to discover inconsistencies. Techniques like **Model Checking** are effective for this.
2. **Code Verification**: Using mathematical proofs to ensure code consistency with its formal specification. Helps in discovering programming and some design errors.

Transformational Development

- The B method is a widely-used formal transformational method, applied in areas like train control systems and avionics software.
- Transforms a formal specification through a series of representations to the final program code.

Advocacy and Criticism

- Advocates argue formal methods lead to **more reliable and safer systems**.
- Critics argue they are **not cost-effective** compared to other validation techniques like inspections and system testing.

Limitations

1. **Specification Inaccuracy**: Formal specifications may not reflect real user requirements.
2. **Proof Errors**: The mathematical proofs can contain errors.
3. **Incorrect Assumptions**: The proofs may assume specific system usage that may not hold true.

Cost and Feasibility

- Formal verification is time-consuming and expensive, requiring specialized tools and mathematical expertise.
- Despite cost concerns, some companies find it cost-effective as it reduces the need for unit testing of components.

Role in Critical Systems

- Formal methods are particularly effective for verifying **critical safety and security components**, even if impractical for large systems.

12.3.3 Model Checking

Overview

- **Model Checking** is an alternative to deductive approaches for formal analysis.
- Focuses on creating a formal state model of a system and validates its correctness using specialized software tools.

Applications

- Initially used for **hardware system designs**.
- Increasingly applied in **critical software systems**, such as NASA's Mars exploration vehicles and Airbus in avionics software.

Tools and Systems

- Numerous tools have been developed including **SPIN**, **SLAM from Microsoft**, and **PRISM**.
- Models are usually expressed in languages specific to the model-checking system, e.g., Promela for SPIN.

Process

1. **Formal Notation**: Desirable system properties are written in formal notation, usually based on temporal logic.
2. **Path Exploration**: The model checker explores all possible state transitions to validate that the property holds.
3. **Confirmation or Counterexample**: If a property holds, the model is confirmed. If not, a counterexample is provided.

Utility

- Particularly useful for **concurrent systems**, which are difficult to test due to sensitivity to timing.

Challenges and Limitations

1. **Model Creation**: Manually creating a model is time-consuming and prone to inaccuracies.
2. **Computational Expense**: Model checking requires exhaustive exploration of all paths, making it computationally expensive.

Future Trends

- Improved algorithms are being developed to make model checking more feasible for large-scale systems.

12.3.4 Static Program Analysis

Overview

- **Static Program Analysis** involves automated software tools that scan the source code to detect potential faults.

- Complements language compiler's error-detection facilities and can be part of the inspection or separate V&V process.

Advantages

- Faster and cheaper than detailed code reviews.
- Does not require specialized notations, making it easier to introduce into a development process.
- Effective for **security checks**, such as buffer overflow vulnerabilities.

Types of Checks

1. **Characteristic Error Checking**: Targets common errors in programming languages like Java or C.
    - Simple and cost-effective; catches 90% of errors with 10 types of characteristic errors.
2. **User-Defined Error Checking**: Users define error patterns based on domain-specific or system-specific knowledge.
3. **Assertion Checking**: The most powerful form, where developers include formal assertions in the code.
    - Analyzer symbolically executes the code to validate these assertions.

Fault Classes Addressed

- **Data faults**: E.g., variables used before initialization, array bound violations.
- **Control faults**: E.g., unreachable code, unconditional branches into loops.
- **Input/Output faults**: E.g., variables output twice without intervening assignment.
- **Interface faults**: E.g., parameter mismatches.
- **Storage management faults**: E.g., unassigned pointers, memory leaks.

Limitations

- **False Positives**: Often flags code sections without errors, requiring additional screening.
- **Incomplete Coverage**: Cannot discover some types of errors that could be identified in program inspections.

Industry Adoption

- Widely used in organizations, including Microsoft, especially where program failures can have serious effects.

12.4 Safety Cases

Overview

- **Safety Cases** are essential in regulated, safety-critical systems to convince external authorities that the system is safe.
- Comprise a set of documents that include system descriptions, logical arguments, and evidence confirming the system's safety.

Purpose

- Provides a convincing and valid argument that a system is adequately safe for its application and environment.

Contents of a Safety Case

1. **System Description**: Overview and critical components of the system.
2. **Safety Requirements**: Derived from the system requirements specification.
3. **Hazard and Risk Analysis**: Documents describing identified hazards and risk mitigation measures.
4. **Design Analysis**: Structured arguments justifying the safety of the design.
5. **Verification and Validation**: Procedures and test results.
6. **Review Reports**: Records of design and safety reviews.
7. **Team Competences**: Evidence of the competence of those involved in safety-related development.
8. **Process QA**: Quality assurance records.
9. **Change Management**: Records of proposed changes and justifications.
10. **Associated Safety Cases**: References to other safety cases that may impact the main safety case.

Industry and Context Specificity

- The structure and detail of safety cases vary by industry and the maturity of the domain (e.g., nuclear vs. medical devices).

Software Safety Case

- A subsection of the main safety case, focusing on software failures and their impact on the overall system.

Cost and Complexity

- Developing safety cases is expensive due to comprehensive validation and record-keeping requirements.

Best Practices

1. **Integrated Development**: The development of a safety case should be tightly integrated with system design to avoid design choices that complicate the safety case.
2. **Regulator Involvement**: Work with regulators early in the development process to understand their expectations for the safety case.

Maintenance

- Changes in software or hardware may require significant portions of the safety case to be rewritten, adding to the cost.

12.4.1 Structured Arguments

Overview

- Decisions on system safety are based on **logical arguments** that link evidence to claims about a system's security and dependability.

Nature of Claims

- Claims can be **absolute** (event X will or will not happen) or **probabilistic** (the probability of occurrence of event Y is 0.n).

Structure of Arguments

- Arguments are usually **claim-based**, meaning they start with a claim about system safety and then present evidence and reasoning to justify that claim.

Example

- **Claim**: The insulin pump will not compute a dose exceeding the maximum safe dose, denoted as maxDose.
- **Evidence**: Test datasets and static analysis reports confirm that the computed dose never exceeds maxDose.
- **Argument**: The evidence demonstrates that the maximum dose of insulin computed is equal to maxDose, justifying the claim.

Redundant and Diverse Evidence

- Utilizing **redundant and diverse processes** for evidence collection increases confidence in the system's safety. If one validation process misses an issue, another is likely to catch it.

Claim Hierarchy

- Safety claims are often organized in a **hierarchical structure**. Lower-level claims must be validated first to support higher-level claims.

Importance

- Structured arguments serve as the logical foundation for the safety case, making it possible to assess with a high level of confidence whether a system is safe.

12.4.2 Software Safety Arguments

Overview

- **Software safety arguments** focus on proving that a program will not reach a potentially unsafe state, rather than proving it works as intended. This makes safety arguments more cost-effective than correctness arguments.

Basic Premise

- The number of system faults leading to safety hazards is usually **significantly less** than the total number of faults in the system. This allows safety assurance to focus specifically on these critical faults.

Steps to Create a Software Safety Argument

1. **Assume Unsafe State**: Start by assuming that an unsafe state, identified in system hazard analysis, can be reached.
2. **Define Predicate**: Write a logical expression that defines this unsafe state.
3. **Analyze System or Program**: Systematically show that all program paths leading to the unsafe state actually contradict the unsafe state predicate.
4. **Validate for All Hazards**: Repeat this analysis for all identified hazards. If all are contradicted, strong evidence exists that the system is safe.

Application Levels

- Safety arguments can be applied at different stages such as **requirements, design models, and code**. Each level aims to validate different aspects like missing safety requirements or unsafe states in design models.

Working Example

- The example uses an **insulin delivery system** to illustrate that the system will not administer an overdose.
    - **Unsafe State Predicate**: currentDose > maxDose
    - **Contradiction**: All program paths lead to a state where currentDose <= maxDose, contradicting the unsafe state predicate.

Importance of Working Backwards

- By working **backwards** from the unsafe state, you can ignore intermediate states and focus solely on the final states that lead to the exit condition for the code. This simplifies the safety argument process.

Key Points to Remember

- This method is effective in proving that the program will not reach an unsafe state under normal conditions, providing strong evidence of system safety.

Summary

- Safety-critical systems are systems whose failure can lead to human injury or death.
- A hazard-driven approach may be used to understand the safety requirements for safety-critical systems.
- You identify potential hazards and decompose them (using methods such as fault tree analysis) to discover their root causes. You then specify requirements to avoid or recover from these problems.
- It is important to have a well-defined, certified process for safety-critical systems development.
- The process should include the identification and monitoring of potential hazards.
- Static analysis is an approach to V & V that examines the source code (or other representation) of a system, looking for errors and anomalies. It allows all parts of a program to be checked, not just those parts that are exercised by system tests.
- Model checking is a formal approach to static analysis that exhaustively checks all states in a system for potential errors.

- Safety and dependability cases collect all of the evidence that demonstrates a system is safe and dependable. Safety cases are required when an external regulator must certify the system before it is used.